A Top-down Presentation of Purdom's Sentence-Generation Algorithm

Brian A. Malloy and James F. Power

NUIM-CS-TR-2005-04

A Top-down Presentation of Purdom's Sentence-Generation Algorithm

Brian A. Malloy
Computer Science Department
Clemson University
Clemson, SC 29634
U.S.A.
malloy@cs.clemson.edu

James F. Power[*]
Department of Computer Science
National University of Ireland
Maynooth, Co. Kildare
Ireland
jpower@cs.nuim.ie

Abstract

In this paper we present a reformulation of a sentence-generation algorithm for context free grammars, originally due to Purdom. We restructure the original algorithm so that it resembles the familiar pattern of a top-down parsing algorithm. We express the algorithm in a declarative style that significantly facilitates its comprehension and implementation.

Purdom's sentence generation algorithm takes a context-free grammar, and generates sentences from the language represented by that grammar [2]. Run to completion, the algorithm will generate a set of sentences such that each rule in the grammar has been used at least once. The original formulation of the algorithm, published in 1972, can be somewhat difficult to understand, since the steps involved are described in an imperative style. In previous work we have reformulated the algorithm in a more procedural style, and implemented it in C++ [1]. This paper takes that process a step further by expressing the algorithm in a more declarative style, yielding a significantly simpler version.

In what follows we describe our reformulation of Purdom's algorithm. The reader is encouraged to consult [1] for more background information. Section 1 presents an overview of our approach, drawing an analogy between the sentence-generation problem and the standard approach to top-down parsing. Section 2 informally describes the crux of the algorithm, which involves choosing the correct rule for a non-terminal at a given stage in the generation process. Section 3 describes *short* and *prev* the main algorithms in this process, in detail.

1. Overview

The overall goal of Purdom's algorithm is to generate a set of strings from the language defined by the grammar that *cover* all the production rules. That is, each production rule is used at least once in generating the set of strings.

Purdom defines a non-terminal as FINISHED if all its production rules (the rules where it occurs on the left-hand-side) have been used. Our goal then is to generate strings until all the grammar's non-terminals are FINISHED.

Purdom's algorithm is thus framed in much the same way as a parsing algorithm except that instead of recognizing tokens in the input string, we are generating tokens to the output string. For simplicity, this paper assumes we are using a top-down algorithm, although Purdom's original algorithm does not insist on this.

Consider the rough outline of the top-down parsing algorithm (e.g. $LL(1)$) using a stack:

1. Push the start symbol on the stack
2. While the stack is non-empty, do:
   (a) Pop the topmost element
   (b) if this element is a terminal, match it with the input
   (c) if it's a non-terminal, choose a rule for the non-terminal, and push the symbols from right-hand-side of the rule onto the stack, in reverse order[1].

Purdom's algorithm follows this template, where instead of matching a symbol with the input, we instead append it to the output. Our version of the generation algorithm is given in Algorithm 1, which generates a single sentence. The only real difference with a parsing algorithm is in line 6, where we print out a symbol instead of matching it against the input.

Algorithm 1 generates a single sentence, and should be run repeatedly until all possible sentences have been generated. In practice, this corresponds to running the algorithm until we record the start symbol as FINISHED.

The crux, of course, is the same as for parsing: how do we choose which production rule to use in line 8? In

[*]James Power is currently on sabbatical at Clemson University, South Carolina, USA.

[1]Reversing the order of the symbols ensures a *leftmost* derivation

Algorithm 1 Top-down generation of a single sentence

```
procedure generate_sentence()
stack ← new Stack()
stack.push($S_0$)
while not stack.isEmpty() do
    $s$ ← stack.pop()
    if $s \in \mathscr{T}$ then
        print s
    else if $s \in \mathscr{N}$ then
        $(s \to \alpha)$ ← choose_rule(s, stack)
        stack.push(reverse($\alpha$))
    end if
end while
```

predictive parsing (such as $LL(1)$) we would look up a table that would tell us which rule generates the next input symbol. With Purdom's algorithm we adopt a must adopt a different approach.

1.1. Notation

In what follows, we regard a grammar as consisting of

- A set $\mathscr{T}$ of terminal symbols
- A set $\mathscr{N}$ of non-terminal symbols, such that $\mathscr{T} \cap \mathscr{N} = \emptyset$
- A distinguished non-terminal $S_0 \in \mathscr{N}$ called the start symbol
- A set of production rules $\mathscr{P}$, of the form $(N \to \alpha)$, where $N \in \mathscr{N}$ and $\alpha \in (\mathscr{T} \cup \mathscr{N})^*$

For a production rule $(N \to \alpha)$, we occasionally refer to N as the let-hand-side of the rule, and the string α as the right-hand-side of the rule.

In giving the types of functions below, we also use *Nat*, the set of natural numbers, and *Bool*, the set of booleans $\{True, False\}$.

2. Choosing a production rule

Since we wish to eventually use all production rules, we might just try to use them in sequence. This is, in fact, the basis of Purdom's algorithm, with a few refinements.

When we've used all the production rules for a non-terminal then it is FINISHED. However, it is often the case that a non-terminal may not have any more of its own productions left to use, but can be used to generate some other non-terminal that is not yet FINISHED. For the purpose of this discussion, we call such a non-terminal *useful*.

Suppose we are looking at a non-terminal N on the stack, and we wish to choose a rule to apply; we proceed as follows:

1. N is not FINISHED
   If there's an unused production rule for N, then select it. If there's more than one unused rule for N, then pick one, using some arbitrary ordering.

2. N is FINISHED, but it is *useful*
   Suppose all the rules for N have been used, but there is some other non-terminal U with unused productions. Then if we can figure out a sequence of rules to apply to N to give us U, we should use these rules.

   Purdom's algorithm pre-computes a table *prev* that gives information on which production rules can generate which non-terminals, and he uses this to create paths of productions from unfinished non-terminals back to finished ones.

3. N is FINISHED and not *useful*
   If there are no unused rules for N, and N can't be used to get to some other non-terminal with unused rules, then we really just want to get N out of the way as simply as possible.

   Purdom precomputes a table called *short* that indicates which production rule produces the shortest derivation for a non-terminal. Thus, if N is not useful, simply choose this production.

There is one slight subtlety with point (2) above. Suppose N can eventually introduce some unfinished non-terminal, U, say. If U is already on the parse stack, then it is possible that step (2) above will take us around in circles, always seeking to introduce an U, but never actually getting around to using it. Thus Purdom marks useful non-terminals that are on the stack as UNSURE.

2.1. The algorithm to choose the next rule

The algorithm makes use of two arrays that are changed as the parse proceeds. The array *once* is used to "line up" the next production rule for a non-terminal, and the array *mark* is used to indicate whether or not a production rule has been used.

$$\begin{array}{rcl} once & : & \mathscr{N} \longrightarrow \{\text{READY}, \text{UNSURE}, \text{FINISHED}\} \cup \mathscr{P} \\ mark & : & \mathscr{P} \longrightarrow Bool \end{array}$$

We assume that both of these are global to all the algorithms defined here. Initially, as shown in Algorithm 2, the *once* array entries are set to READY, and the *once* array entries are set to *False*.

The function *choose_rule*, called on line 9 of Algorithm 1, is defined in Algorithm 3. If the *once* array doesn't hold a rule for the non-terminal, then we call *calc_paths* to pick one. If there's an unused or useful rule for the non-terminal

Algorithm 2 Initialize *mark* and *once* the two tables that keep track of the state during sentence generation

```
 1: procedure init_dynamic_tables()
 2: for all $N \in \mathscr{N}$ do
 3:     $once[N] \leftarrow$ READY
 4: end for
 5: for all $(N \rightarrow \alpha) \in \mathscr{P}$ do
 6:     $mark[(N \rightarrow \alpha)] \leftarrow$ False
 7: end for
```

Algorithm 3 The main loop that chooses a new rule for a non-terminal.

```
 1: function choose_rule ($N : \mathscr{N}$, $pStack : Stack$) : $\mathscr{P}$
 2: if $once[N] =$ READY or $once[N] =$ UNSURE then
 3:     calc_paths($N, pStack$) // Steps 4-7
 4: end if
 5: // Work out the rule we'll choose:
 6: if $once[N] \in \mathscr{P}$ then // 8. Get production
 7:     $(N \rightarrow \alpha) \leftarrow once[N]$
 8:     $once[N] \leftarrow$ READY
 9: else // 9. Take shortest
10:     $(N \rightarrow \alpha) \leftarrow short[N]$
11:     $mark[(N \rightarrow \alpha)] \leftarrow$ True
12: end if
13: return $(N \rightarrow \alpha)$
```

it will be assigned to *once*$[N]$, and we'll use it (lines 6-8); this is step 8 of Purdom's algorithm. The *once* entry for N is set back to READY in line 8 so that a new rule will be chosen the next time around. If a useful or unused rule could not be found for the non-terminal we just use the shortest possible rule and mark that rule as used (line 9-11); this is Purdom's step 9.

The purpose of *calc_paths*, called on line 3 of Algorithm 3, and defined in Algorithm 4, is to line up the next production rule. It does this in three phases, corresponding to steps 4, 5 and 7 of Purdom's original algorithm.

First, we look at all *unused* production rules, and if the corresponding non-terminal is marked READY or UNSURE, then we line up this unused rule as the next one to use (lines 3-10). Note that we mark a rule (line 7) as used as soon as it's lined up in the *once* array; we don't have to wait for *choose_rule* to return it.

Next, having selected the rules we want to use for some of the non-terminals, we see if we can line up the other non-terminals to lead us to these rules. Lines 12-18 look for non-terminals other that the start symbol whose *once* value is a production rule, and calls *setup_paths*, defined in Algorithm 5, for these.

Finally, if any non-terminal still has a once value of READY after all this, then it is marked as FINISHED (lines 20-24).

Algorithm 4 We've decided some non-terminal is ready for a new rule, so find one.

```
 1: procedure calc_paths($pStack : Stack$)
 2: // 4. Link to new production:
 3: for all $(N \rightarrow \alpha) \in items$ do
 4:     if not $mark[(N \rightarrow \alpha)]$ then
 5:         if $once[N] =$ READY or $once[N] =$ UNSURE then
 6:             $once[N] \leftarrow (N \rightarrow \alpha)$
 7:             $mark[(N \rightarrow \alpha)] =$ True
 8:         end if
 9:     end if
10: end for
11: // 5. Find paths
12: for all $(N, status) \in once$ do
13:     if $N = S_0$ then
14:         continue
15:     else if $status \in \mathscr{P}$ then
16:         setup_paths($N, pStack$)
17:     end if
18: end for
19: // 7. Mark finished
20: for all $(N, status) \in once$ do
21:     if $status =$ READY then
22:         $once[N] \leftarrow$ FINISHED
23:     end if
24: end for
```

The function *setup_paths* takes a non-terminal and figures out what other non-terminals are *useful* in deriving it, and is defined in Algorithm 5. It uses the *prev* array, defined later, to walk back up through the grammar from some non-terminal N_i towards the start symbol S_0. Any non-terminal it encounters along the way that doesn't already have a rule lined up in its *once* entry, and that isn't on the parse stack, has it's *once* value assigned to the rule retrieved from *prev* (lines 8-10). A useful-looking non-terminal that *is* on the parse stack is given the value UNSURE (line 12).

This is the essence of Purdom's algorithm. The only remaining parts are the static pre-computation of the two arrays, *short* and *prev* used on line 10 of Algorithm 3 and line 5 of Algorithm 5 respectively.

3. Calculating *short* and *prev*

The two arrays *short* and *prev* used in Purdom's algorithm can be usefully compared to the functions *FIRST* and *FOLLOW* used in top-down predictive parsing (such as $LL(1)$). All can be derived statically from the grammar, represented in tabular form, and then accessed quickly during the parse.

Algorithm 5 Work out paths to unused productions by following *prev* back up through the grammar, starting at N_i

```
procedure setup_paths($N_i : \mathcal{N}$, $pStack : Stack$)
// 6. Set up paths
$N_j \leftarrow N_i$
while $N_j \neq S_0$ do
    $(N_j \rightarrow \alpha) \leftarrow prev[N_j]$
    if $once[N_j] \in \mathcal{P}$ then
        return
    else if $N_i \notin pStack$ then
        $once[N_j] \leftarrow (N_j \rightarrow \alpha)$
        $mark[(N_j \rightarrow \alpha)] \leftarrow$ True
    else
        $once[N_j] \leftarrow$ UNSURE
    end if
end while
```

Further, both *short* and $FIRST$ are instances of *bottom up* grammar flow analysis problems, in the sense of Wilhelm & Maurer [3]. That is, both are calculated for a non-terminal by examining rules where the non-terminal appears on the left-hand-side, and applying some function to the symbols on the right-hand-side. Thus, values of *short* (and $FIRST$) work their way up through the grammar from the terminal symbols, back towards the start symbol.

Similarly, both *prev* and $FOLLOW$ are instances of *top-down* grammar flow analysis problems. For a non-terminal N, we calculate either *prev* or $FOLLOW$ by looking for rules where N occurs on the right hand side, and transferring a value from the non-terminal on the left-hand-side of the rule. Thus values of *prev* (and $FOLLOW$) flow down through the grammar from the start symbol. The relationship to grammar flow analysis problems is discussed further in Appendix A.

The array *short* maps a non-terminal to the production rule that uses it as simply as possible. The array *prev* maps a non-terminal to a rule that will introduce it into a parse. Both are calculated statically, before sentence generation, and are not altered during sentence generation.

$$\begin{aligned} short &: \mathcal{N} \longrightarrow \mathcal{P} \\ prev &: \mathcal{N} \longrightarrow \mathcal{P} \end{aligned}$$

Purdom uses two intermediate structures $rlen$ and $dlen$ that are local to Algorithms 6 through 9. Both of these will contain natural numbers, representing the size of strings in the (shortest) parse.

$$\begin{aligned} rlen &: \mathcal{P} \longrightarrow Nat \\ dlen &: \mathcal{N} \longrightarrow Nat \end{aligned}$$

The basic structure of the algorithm that uses $rlen$ and $dlen$ in the calculation of *short* and *prev* is given in Algorithm 6. Initially, all $rlen$ and $dlen$ are set to infinity, and then we use a fix-point algorithm to calculate their proper values, which calculates *short* and *prev* as a side-effect. The procedure *calc_rlen* calculates both $rlen$ and *short*, while *calc_dlen* uses $rlen$ in the calculation of both $dlen$ and *prev*.

Algorithm 6 The main algorithm for calculating the two static tables, *short* and *prev*. Both $dlen$ and $rlen$ are local to this process, and may be discarded after this algorithm finishes.

```
procedure calc_static_tables()
// Initialise
rlen, short, dlen, prev $\leftarrow \emptyset, \emptyset, \emptyset, \emptyset$
for all $(N \rightarrow \alpha) \in \mathcal{P}$ do
    $rlen[(N \rightarrow \alpha)] \leftarrow \infty$
end for
for all $N \in \mathcal{N}$ do
    $dlen[N] \leftarrow \infty$
end for
// Iterate until fix-point
repeat
    changed $\leftarrow$ calc_rlen(rlen, short)
until not changed
repeat
    changed $\leftarrow$ calc_dlen(rlen, dlen, prev)
until not changed
// short and prev are now ready for use
```

Algorithm 7 Calculate *slen* based on the current value of $rlen$. Returns 1 for terminals, or the min $rlen$ value for non-terminals.

```
1: function slen($s : (\mathcal{T} \cup \mathcal{N})$) : Bool
2: if $s \in \mathcal{T}$ then
3:     return 1
4: else if $s \in \mathcal{N}$ then
5:     return $min\{(s \rightarrow \alpha) \in \mathcal{P} \cdot rlen[(s \rightarrow \alpha)]\}$
6: end if
```

3.1. Calculating $rlen$ and *short*

The array $rlen$ gives the shortest string that can be generated by starting with a given production rule. The function *slen*, defined in Algorithm 7, effectively generalizes this to terminals and non-terminals. The length of the string generated from a terminal symbol is always 1 (lines 2-3), and the length for a non-terminal is the minimum $rlen$ for all rules for that non-terminal (lines 4-5).

To actually calculate $rlen$, we iteratively pull the *slen* values up through the grammar from the terminals, as shown in Algorithm 8. The algorithm looks at each production rule once, to see if its $rlen$ value needs to be updated. Given a rule of the form $(N \rightarrow \alpha)$, we calculate the

Algorithm 8 Iterate (once) through the rules, pulling values of *rlen* upwards. Return true or false depending on whether anything changed.

```
 1: function calc_rlen(rlen, short) : Bool
 2: changed ← False
 3: for all $(N \to \alpha) \in \mathscr{P}$ do
 4:     $sum \leftarrow 1 + \sum_{\alpha_i \in \alpha} slen(\alpha_i)$
 5:     if $sum < rlen[(N \to \alpha)]$ then // New min for rule
 6:         $rlen[(N \to \alpha)] \leftarrow sum$
 7:         changed ← True
 8:         if $sum < slen(N)$ then // New min for non-t too
 9:             $short[N] \leftarrow (N \to \alpha)$
10:         end if
11:     end if
12: end for
13: return changed
```

Algorithm 9 Iterate (once) through the rules, moving values of *dlen* downwards. Return true or false depending on whether anything changed.

```
 1: function calc_dlen(rlen, dlen, prev) : Bool
 2: changed ← False
 3: $dlen[S_0] \leftarrow slen(S_0)$
 4: for all $(N \to \alpha) \in \mathscr{P}$ do
 5:     if $dlen[N] = \infty$ then
 6:         continue
 7:     end if
 8:     // $c$ is the min string length if we use $(N \to \alpha)$
 9:     $c \leftarrow (dlen[N] + rlen[(N \to \alpha)] - slen(N))$
10:     // $s$ is each non-terminal on the RHS of the rule:
11:     for all $s \in \alpha$ do
12:         if $s \in \mathscr{N}$ and $c < dlen[s]$ then // New min
13:             $dlen[s] \leftarrow c$
14:             $prev[s] \leftarrow (N \to \alpha)$
15:             changed ← True
16:         end if
17:     end for
18: end for
19: return changed
```

value of *rlen* for N based on the values for the symbols in α. The potential "new" *rlen* value is calculated in line 4 as the sum of the *slen* for each symbol on the right-hand-side, plus one, for the extra rule. Lines 5-11 simply update the value of *rlen* for the rule, and *short* for the non-terminal, if a new minimum has been found.

3.2. Using *rlen* to calculating *dlen* and *prev*

The final algorithm uses *rlen* to calculate both *dlen* and *prev* and is shown in Algorithm 9. The array *dlen* maps a non-terminal to the length of the shortest string that uses that non-terminal in its production. Similarly to *calc_rlen*, we are iterating through the grammar rules, re-calculating the value of *dlen*, looking for a new minimum. However, this time, given a rule of the form $(N \to \alpha)$, we are attempting to transfer the *dlen* value from N to the non-terminals contained in α.

It helps to think of *dlen* as carrying the extra "cost" of choosing rules to deliberately introduce a non-terminal, instead of just selecting rules that generate the shortest string. Since S_0 is always used, its cost is the lowest, i.e. its *slen* value (line 3). For all other non-terminals, we must iteratively plot a path from S_0 to that non-terminal, picking up any extra cost along the way. The cost calculation is shown in the assignment to c in line 9 of Algorithm 9: it's the cost of getting here from the start symbol, $dlen[N]$, plus the cost of using this rule instead of the shortest, $(rlen[(N \to \alpha)] - slen(N))$.

4. Implementation

We have implemented the above algorithm in the Python programming language. Most of the algorithms presented above are almost directly translatable into Python, and the only significant additional code needed was a class to represent grammars, and to parse yacc-like grammar specifications, for convenience. The total implementation is achieved in slightly less than 500 lines of Python code, a significant simplification of our previous version of Purdom's algorithm in approximately 3,000 lines of C++ code [1].

We have used the implementation to generate sentences for several example grammars, as well as programming language grammars such as C, C++ and Java. However, we must repeat here the caveat noted in our previous work, namely, that the sentences generated for programming language grammars may not always be correct programs. This is due to a tendency to under-specify programming language syntax in a grammar, and the lack of context-sensitive information in the grammar. The reader should consult [1, §IV] for more details.

References

[1] Brian A. Malloy and James F. Power. An interpretation of purdom's algorithm for automatic generation of test cases. In *1st Annual International Conference on Computer and Information Science*, Orlando, Florida, USA, October 3-5 2001.

[2] P. Purdom. A sentence generator for testing parsers. *BIT*, 12:366–375, April 1972.

[3] R. Wilhelm and D. Maurer. *Compiler Design*. Addison Wesley, 1995.

A. Grammar Flow Analysis

Wilhelm & Maurer use the generalized structure of a grammar flow analysis (GFA) problem to describe iterative algorithms over grammars [3, §8.2.4]. In particular, $FIRST$ and non-terminal productivity can be characterized as instances of *bottom-up* GFA problems, while $FOLLOW$ non-terminal reachability can be characterized as instances of *top-down* GFA problems. In this section we re-cast the $rlen$ and $slen$ algorithms as GFA problems.

We follow the notation of [3], where a production rule p is regarded as an array of symbols, with $p[0]$ being the non-terminal on the left-hand-side, and $p[1],\ldots$ representing the symbols on the right-hand-side. The number of symbols on the right-hand-side of a production rule p is denoted n_p.

A.1. $rlen$ as a bottom-up GFA problem

Formally, a bottom-up GFA problem consists of:

- A domain $D\uparrow$, the set of information associated with non-terminals
- A transfer function $F_p\uparrow: D\uparrow^{n_p} \rightarrow D\uparrow$ for each production rule $p \in \mathscr{P}$ that moves information from the rule's right-hand-side to the non terminal on its left-hand side.
- A combination function $\nabla\uparrow: 2^{D\uparrow} \rightarrow D\uparrow$ that combines the information from several rules for a non-terminal into the value in $D\uparrow$ for that non-terminal.

Technically $rlen$ operates on a grammar rule, but blurring the distinction between $rlen$ and $slen$, we can formulate $rlen$ as an instance of a bottom-up GFA problem as follows:

- The domain $D\uparrow$ is the set of natural numbers:

$$D\uparrow = Nat$$

- The transfer function for some production rule p, $F_p\uparrow$: $Nat^{n_p} \rightarrow Nat$ is defined on line 4 of Algorithm 8, i.e.

$$F_p\uparrow = 1 + \sum_{i=1}^{n_p} slen(p[i])$$

- The combination function $\nabla\uparrow$: $2^{Nat} \rightarrow Nat$ simply chooses the minimum of the $rlen$ values, i.e.

$$\nabla\uparrow = min$$

Thus we can define the $rlen$ value of a non-terminal $X \in \mathscr{N}$ as being:

$$\boxed{rlen(X) = min\{1 + \sum_{i=1}^{n_p} slen(p[i]) \mid p[0] = X\}}$$

A.2. $slen$ as a top-down GFA problem

Formally, a top-down GFA problem consists of:

- A domain $D\downarrow$
- n_p transfer functions for each production rule $p \in \mathscr{P}$, $F_{p,i}\downarrow: D\downarrow \rightarrow D\downarrow$ to move information from the non-terminal on the left-hand-side of the rule to each symbol on the right-hand-side of the rule.
- A combination function $\nabla\downarrow$: $2^{D\downarrow} \rightarrow D\downarrow$ that combines the information from several non-terminal occurrences to the value in $D\downarrow$ for that non-terminal.
- A value $I_0 : D\downarrow$ for the start symbol S_0.

We can formulate $dlen$ as an instance of this as follows:

- The domain $D\downarrow$ is the set of natural numbers:

$$D\downarrow = Nat$$

- The transfer function for the i^{th} symbol on the right-hand-side of some production rule p, $F_{p,i}\downarrow: Nat \rightarrow Nat$ is defined on line 9 of Algorithm 9, i.e.

$$F_{p,i}\downarrow = (dlen(p[0]) + rlen(p) - slen(p[0]))$$

- The combination function $\nabla\downarrow$: $2^{Nat} \rightarrow Nat$ simply chooses the minimum of the $dlen$ values, i.e.

$$\nabla\downarrow = min$$

- The initial value of $dlen$ for the start symbol is just its $slen$ value, as defined on line 3 of Algorithm 9

$$I_0 = slen(S_0)$$

Thus we can define the $dlen$ value of a non-terminal $X \in \mathscr{N}$ as being:

$$\boxed{\begin{array}{rcl} dlen(S_0) & = & slen(S_0) \\ dlen(X) & = & min\{(dlen(p[0]) + rlen(p) - slen(p[0])) \\ & & \mid p[i] = X, 1 \le i \le n_p\} \text{for } X \in (\mathscr{N} - S_0) \end{array}}$$